jambonz handles 10× LiveKit's call volume

A reproducible benchmark of self-hosted voice AI concurrency, measured on identical hardware

Published August 2026

1. Executive Summary

We are in an age where Voice AI technology is moving from small scale Proof-of-Concepts to scalable production deployments, and with that shift comes an old-fashioned engineering question: Just how many concurrent voice-agent calls can one server carry?

We built an open-source test harness[1] (a SIP/RTP load generator plus deterministic mocks of the speech and language-model vendors) and used it to measure the sustainable capacity of two self-hosted voice-agent orchestration platforms:

- **jambonz**: a SIP/telephony-first Voice AI orchestration platform whose media plane is a single shared, compiled media-server process.
- **LiveKit**: a widely used WebRTC-first platform, tested with its Agents framework and its SIP gateway, in which each call is served by its own agent job subprocess.

1.1 jambonz sustains 10x more concurrent voice-agent sessions than LiveKit on identical hardware

On identical 8-vCPU hardware, running the same eight-turn conversation against the same mocked vendors, jambonz scales 10x further than LiveKit.

jambonz sustained ~250 concurrent voice-agent sessions and LiveKit sustained ~25, both measured at the same success threshold defined in §3.1 (fewer than 0.5% failed calls and p95 turn latency within 25% of baseline). Normalised for hardware, that is approximately 31 versus 3 sessions per vCPU.

Configuration	Platform vCPU	Sustainable sessions	Sessions / vCPU
jambonz — single node	8	250	31.3
LiveKit — single node	8	25	3.1
jambonz — clustered (medium reference)	16	450	28.1
LiveKit — 2 agent servers (its reference topology)	16	35	2.2
LiveKit — 4 agent servers	24	70	2.9

Table 1. Sustainable capacity at the quality threshold defined in §3.1 — fewer than 0.5% failed calls and p95 turn latency within 25% of the platform's unloaded baseline. Rows are paired by hardware: 8 vCPU against 8 vCPU, then 16 against 16. LiveKit was additionally tested in four single-node variants (§5.2); all landed at approximately 25.

The comparison holds in both directions. On one 8-vCPU instance the ratio is tenfold. Comparing each platform in its own clustered reference topology on 16 vCPU, jambonz sustained 450 sessions against LiveKit's 35, a ratio of roughly thirteen to one, in the configuration LiveKit's own documentation and community recommended to us.

1.2 Why the result is credible

Three properties of the data matter more than the size of the ratio:

1. **It is not a hardware artifact:** LiveKit measured between 2.2 and 3.1 sessions per vCPU in every one of seven configurations we tested: single node with default settings, single node tuned per Livekit recommendations, with native turn detection as well as STT-based turn detection, with local voice-activity detection removed, and in two- and four-node clusters. Giving LiveKit more hardware increased its absolute capacity proportionally but did not improve its efficiency.
2. **It is not an artifact of turn-detection choice:** We re-ran both platforms with identical, off-box turn detection (Deepgram Flux STT protocol), which removes the turn detector from the comparison entirely. LiveKit's ceiling did not move. Its latency improved markedly, but its capacity did not.
3. **We can explain the mechanism:** Per-process CPU sampling shows LiveKit spending roughly 0.11 CPU cores per concurrent session inside its per-call agent processes, near-constant across load and topology, against roughly 0.014 cores per session in jambonz's shared media process, an eight-fold difference in the cost of carrying one call.

1.3 Latency

Capacity might be the headline, but response latency is the obvious follow-up question, so we measured that as well. With identical off-box turn detection at matched load, jambonz's median turn latency was approximately 96 ms lower (bootstrap 95% CI [–99, –93] ms). The effect is modest in absolute terms (about 6% of end-to-end response time) but exceptionally reproducible: the per-run ranges of the two platforms did not overlap across six independent runs spanning three different LiveKit topologies.

1.4 What we did to make LiveKit fast

A vendor-run benchmark is worth nothing if the competitor was left misconfigured, so we treated tuning LiveKit as part of the work:

- We diagnosed and fixed a configuration issue that initially capped LiveKit at roughly 14 sessions.
- We opened up its worker admission thresholds, then reverted them on community advice after confirming the defaults performed better.
- We moved turn detection off the box, and removed the last remaining per-call inference workload.
- We rebuilt the deployment into LiveKit's own documented reference topology with dedicated agent servers, then doubled the agent tier.
- We published our configuration to the LiveKit community, asked what we got wrong, and re-tested against every recommendation we received.

Independently, LiveKit's own deployment documentation rates an agent server of this size at *"4 cores and 8GB … 10-25 concurrent jobs,"* which is consistent with what we measured.

1.5 Scope and limitations

LiveKit is a substantially broader platform than jambonz. It does video, multi-party SFU rooms, better WebRTC support, and client SDKs across many languages. Our study measures exactly one thing: the density of telephony-originated voice-agent sessions on self-hosted hardware. Readers should not generalise beyond that.

This is also not a cost-of-ownership study. Note that the speech and language-model vendor charges, which are identical for both platforms and which in production would typically exceed infrastructure cost, are outside its scope.

1.6 Which platform for which workload?

Capacity matters only when you need it. No platform wins outright here.

1. **Below roughly 100 concurrent sessions, LiveKit is a perfectly sound choice.** At that scale its per-session cost amounts to a handful of small nodes, well within what any team can operate, and the things it does that jambonz does not (video, multi-party rooms, browser and mobile client SDKs) are likely to matter far more to a project than the compute it consumes. Nothing in this paper argues against choosing it for that range, and our measurements show it scaling cleanly and predictably within it.
2. **Above that point the arithmetic begins to dominate the decision.** At 1,000 concurrent sessions the same workload is roughly four 8-vCPU instances on jambonz against roughly forty on LiveKit, and the operational weight of the larger fleet (deployment, monitoring, failure domains, capacity planning) compounds along with the bill. If sustained concurrency in the high hundreds or thousands is a requirement, that is where a shared media plane earns its keep, and where we would encourage a team to evaluate jambonz.
3. **Stated as a rule of thumb:** if you are sizing for tens of concurrent calls, pick on features. If you are sizing for hundreds or thousands, pick on architecture, and consider jambonz.

1.7 Reproducibility

The load generator, the vendor mocks, every configuration file, the provisioning scripts and the complete measurement data set are published on github. We invite you to reproduce the test, tweak our configuration if desired, and publish their results.

> Ref: https://github.com/jambonz/jambonz-livekit-load-testing-benchmarks

2. Background

This section describes each platform's architecture and states why we expected those architectures to differ in session density.

2.1 The workload: what a Voice AI agent does

A Voice AI agent is a pipeline. Audio arrives from the caller as RTP. Speech-to-text produces a transcript. Some mechanism decides when the caller has finished speaking (the turn-detection problem). The transcript goes to a language model, whose response is streamed to a text-to-speech service, whose audio is streamed back to the caller. If the caller interrupts, playback must stop and the cycle restarts.

Most of the heavy lifting in that pipeline is performed by the AI models provided by external vendors. What the platform provides is the harness for media handling and orchestration: terminating the call, moving audio, maintaining conversational state, and deciding when to speak.

Platform capacity is therefore a question about the **cost of media handling and orchestration per concurrent call**, which is what this benchmark attempts to isolate by holding the vendors constant.

2.2 LiveKit

LiveKit is a real-time communications platform built around a WebRTC selective forwarding unit (SFU). Its original and primary domain is browser and mobile real-time media (video conferencing, livestreaming, and multi-party audio rooms) and it is widely adopted for those purposes. Two additional components bring it into the telephony voice-agent space:

- **Livekit-sip**: a SIP gateway that terminates SIP calls and bridges them into the SFU as WebRTC participants.
- **Livekit-agents**: a Python framework in which an "agent" joins a room as a participant. A worker process registers with the server and, for each dispatched job, **forks a dedicated subprocess** that runs that call's STT, LLM, TTS and turn-detection pipeline.

The process-per-call model is a deliberate engineering choice. It gives strong fault isolation (a crashing agent takes down one call, not the server) and it sidesteps Python's global interpreter lock, which would otherwise serialise concurrent CPU-bound work such as local voice-activity detection.

LiveKit's documentation is explicit about the resulting sizing, recommending four cores and 8 GB per agent server for 10 to 25 concurrent jobs, and its scaling story is horizontal: add agent servers, and the server distributes jobs across them.

2.3 jambonz

jambonz is a voice AI orchestration platform with CPaaS roots: built SIP-first, for carrier-facing telephony. Its components divide along different lines:

- **drachtio**: a SIP server handling signalling; the SBC tier also runs an RTP proxy for media relay.
- **The jambonz media server**: a single compiled process per host that owns media for all calls on that host: RTP handling, audio codecs, playback, recording, and the streaming connections to speech vendors. Sessions are lightweight objects inside one long-lived process rather than processes in their own right.
- **Feature-server:** a Node.js orchestrator that executes the application's call flow, including the agent verb that wires STT, LLM, TTS and turn detection together and manages conversational state.
- **The customer application**: a thin webhook or WebSocket service that returns instructions. In this benchmark it is a small Node application, and (as §7 records) it became a bottleneck before the platform did, which is itself a deployment lesson.

2.4 Hypothesis

A process-per-call runtime pays a fixed cost for every concurrent session before any useful work is done: interpreter startup, model loading into that process, inter-process communication with the parent worker, and an additional entry in the operating system's scheduling set.

A shared media plane amortises those costs across all sessions on the host, paying them once per process rather than once per call.

We therefore predicted two things, both testable in §5:

- **Per-session CPU cost would differ by roughly an order of magnitude**, with the per-call process model the more expensive.
- **LiveKit's ceiling would be insensitive to turn-detector choice**, because if the dominant cost is the per-call process itself, then moving inference out of that process should not raise the ceiling.

3. Methodology

3.1 The capacity metric and the performance threshold

We report *sustainable concurrent sessions*, defined against an explicit performance threshold. A "session" is one inbound SIP call carrying bidirectional RTP through a complete eight-turn conversation, in which the agent responds to every turn. A session counts only if the call completes: setup succeeds and every expected agent response arrives within its timeout.

Sustainable capacity is the highest tested concurrency at which fewer than 0.5% of calls fail and p95 turn latency stays within 25% of that same platform's unloaded baseline.

vCPU counts are allocated vCPUs across the entire deployment, including orchestration, database and cache nodes that carry no media.

3.2 Workload

Every call runs the same scripted conversation:

An eight-turn airline flight-change dialogue of roughly three minutes, spoken by a synthetic caller as G.711 audio over real SIP and RTP.

The caller side is reproduced verbatim below. The agent's replies are summarised with their word counts, which is what determines how long the agent holds the floor.

#	Caller utterance (verbatim)	Agent reply
	(call answered)	Greeting — 18 w
1	Hi, I need to make a change to an existing reservation I have.	Asks departure city and date — 25 w
2	It's a flight from Chicago to Denver next Monday morning.	Confirms the booking, asks what change — 28 w
3	I'd like to move it to **[800 ms]** um **[700 ms]** Thursday afternoon instead, if there's availability.	Offers three Thursday afternoon options — 31 w
4	Do any of those options include a direct flight?	Two are direct, one has a layover — 27 w
5	The two fifteen departure works. What would the fare difference be?	Quotes the $42 difference, asks to confirm — 29 w
6	Okay, that's fine. Go ahead and **[900 ms]** uh **[600 ms]** use the card I have on file.	Confirms the rebooking and the charge — 39 w
7	No, that's everything. Can you email me the updated itinerary?	Confirms the itinerary was sent — 24 w
8	Great, thanks so much for your help. Goodbye.	Closing — 15 w

Table 2. The scripted conversation. Bracketed values are silent pauses inside a single caller utterance. Agent replies are paraphrased. Word counts are exact.

The hesitation pauses are intended to represent the reality of the real-world**.** They are 600, 700, 800 and 900 ms. A fixed silence timer must exceed 900 ms to ride out all four. At that setting, it adds the same 900 ms to the end of every *normal* turn, where the caller has finished. **There is no timeout value that is both patient enough for turns 3 and 6 and fast enough for the other six.** This is why production turn detection is a model problem rather than a timeout problem, and it is why the turn detectors under test here are doing real work instead of being trivially satisfied.

Two further properties are deliberate. The caller hangs up after their final utterance, as a real caller would, which avoids triggering idle-prompt behaviour that would contaminate the measurement. And the caller audio is pre-rendered once into fixed G.711 fixtures, so every call on every platform transmits byte-identical audio. The same voiced spans, the same pause lengths, in the same order.

3.3 Deterministic vendor mocks

Both platforms are pointed at the same mock vendor host, which implements four protocols with fixed, injected latencies:

Mock service	Protocol	Injected latency
Speech-to-text	Deepgram Nova-3 STT streaming	interim results every 250 ms
Speech-to-text with turn events	Deepgram Flux STT	EndOfTurn 350 ms after end of speech; EagerEndOfTurn 200 ms earlier
Text-to-speech	Deepgram streaming TTS	first audio 150 ms after flush
Language model	OpenAI-compatible	first token 400 ms; 60 tokens/second

Table 3. Mocked vendor services. Latencies are modelled on published vendor figures and are identical for both platforms.

No traffic reaches any speech or language-model vendor, and no vendor charges are incurred. Where vendor names appear in configuration ("deepgram", "openai") they select a *protocol dialect*, and the endpoint is overridden to the mock host. Holding the vendors fixed and deterministic is what allows differences between the platforms to be attributed to the platforms.

3.4 Turn detection as a controlled variable

Turn detection is the one pipeline component the platforms implement differently by default, and it is CPU-intensive, so it would otherwise confound the comparison.

We therefore tested each platform twice:

- **With its own native detector**: Krisp's v3 turn-taking model on jambonz; the LK proprietary transformer-based semantic turn detector on LiveKit. This is what each platform recommends, and it is the configuration a normal user would deploy.
- **With identical off-box turn detection**: Deepgram Flux, whose turn events come from the STT service itself, mocked identically for both. In this configuration neither platform performs turn-detection inference locally, so the comparison isolates media and runtime architecture.

3.5 Environment and instrumentation

All systems ran on AWS Graviton instances (c7g and c7gn families) in a single availability zone, in one VPC, to keep network conditions uniform and avoid cross-zone media. The load generator and the vendor mocks ran on a separate host from every system under test.

Resource data was sampled every two seconds on every node, capturing whole-machine CPU broken out by mode (user, system, iowait, softirq), load average, memory, network packet rates, and, critically for §5.5, per-process CPU and proportional-set-size memory for every platform component. Per-process CPU is computed from per-PID deltas so that pools of short-lived processes are accounted for correctly, and memory uses PSS so that shared pages in forked workers are not double-counted.

- **Per-process figures cover userspace work only:** Where a component's cost does not appear against a process, media relay on the SBC tier being the notable case, we take it from whole-node CPU and interface throughput instead, and label it as derived. Reading such a tier from its process CPU alone would understate it.
- **Validity check:** Every run records load-generator and mock-host utilisation. Across all runs reported here the harness never exceeded roughly 14% CPU, confirming that the measured ceilings belong to the systems under test and not to the test rig.

3.6 Ladder procedure

Capacity is located by a stepped ladder. Each step holds a fixed concurrency for a sustained period, originating enough calls to cycle the population several times, then records completion and latency statistics. A step exceeding 25% failures triggers a safety stop. Where the transition from clean to failing spans a wide interval, a second finer-grained ladder brackets the knee.

3.7 Statistical treatment of latency

Turn latency is measured on the wire by the load generator as the interval between the caller's last voiced audio frame and the first frame of the agent's reply, corrected for the silence tail of the caller's audio fixture.

Because turns within a call share an environment and are therefore correlated, pooling all turns would understate variance and inflate significance. We use each call's mean latency as the unit of analysis, compare distributions with the Mann-Whitney U test, report Cliff's delta as a robust effect size, and

bootstrap confidence intervals for the difference in medians. We report effect size and distributional overlap rather than leaning on p-values, which are uninformative at this sample size.

3.8 Cost derivation

Cost is derived from capacity rather than measured separately:

cost per session = vCPU-hour price ÷ sessions per vCPU

Prices are on-demand list prices for the specific instance types used, in the stated region, on the stated date. **We do not assume negotiated, reserved, or spot discounts.** Those reduce both platforms proportionally and so do not affect the ratio. Within the Graviton c7g family, price per vCPU-hour is uniform across instance sizes, which makes sessions-per-vCPU convert cleanly into cost.

4. Test Configurations

Eleven configurations were measured. This section describes the deployment architectures; §5 reports what each one produced.

4.1 Common test rig

Every configuration shares the same load-generation and mock tier, so that only the system under test changes:

- **Load generator**: a purpose-built SIP user agent that originates real calls, streams pre-rendered G.711 caller audio, and detects the agent's replies by energy-based voice-activity detection on the returning RTP. It measures turn latency from the wire, independent of any platform-reported metric. A single process comfortably sustains several hundred concurrent calls.
- **Mock vendor host**: one service exposing the four mocked vendor protocols from §3.3, plus a metrics endpoint. Script-aware and audio-aware: it detects the caller's voiced spans in the incoming audio and paces scripted transcripts against them, so both platforms' turn-taking code paths are exercised.
- **Placement**: both run on one host, in the same availability zone and VPC as the system under test, on 8 vCPU. Utilisation is recorded as a validity check on every run.

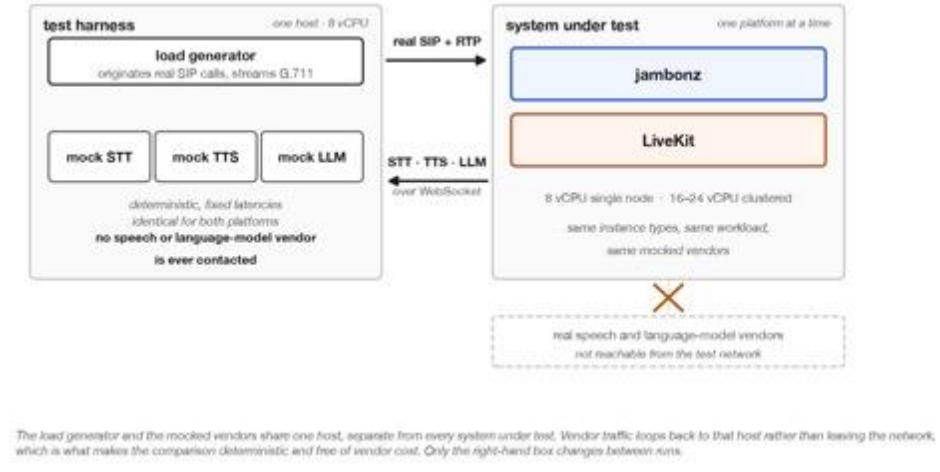


Figure 1. The test rig. The load generator and the mocked speech and language-model services share one host, separate from every system under test; vendor traffic loops back to that host rather than leaving the network.

4.2 Single-node configurations

The single-node tests are the cleanest comparison in the study: one instance type, one machine, everything for a platform co-located on it.

	jambonz single node	LiveKit single node
Instance	c7g.2xlarge (8 vCPU / 15.3 GiB, Graviton)	c7g.2xlarge (8 vCPU / 15.3 GiB, Graviton)
Signalling	drachtio (SIP)	livekit-sip (SIP→WebRTC bridge)
Media	jambonz media server (one shared process) + RTP proxy	livekit-server SFU + per-call WebRTC transports
Orchestration	feature-server (Node.js, 8 instances)	livekit-agents worker
Per call	a session object inside the shared media process	a forked Python job subprocess
Customer code	Node WebSocket app (clustered, 4 instances)	the agent script itself
State store	Redis (local)	Redis (local)

Table 4. Single-node deployments. Both platforms occupy one identically-sized instance; only the internal decomposition differs.

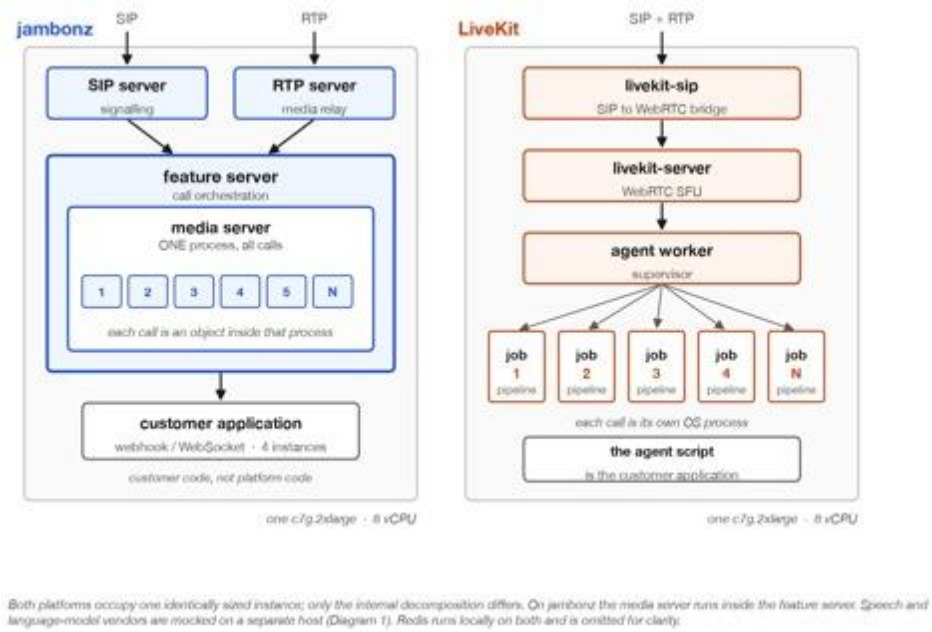


Figure 2. Single-node architectures. Both platforms occupy one identically sized instance; only the internal decomposition differs. On jambonz each call is an object inside one shared media-server process; on LiveKit each call is its own operating-system process.

4.3 Clustered configurations

4.3.1 LiveKit reference topology

LiveKit's documentation prescribes a tiered deployment: agents on their own nodes, separated from the SFU, with jobs distributed across a worker pool and scaling achieved by adding agent servers. Its community independently recommended the same thing to us: that co-locating the SFU and the agent

runtime on one box causes CPU contention, and that moving agent workers onto their own nodes is the single largest available lever.

We built exactly that.

Node	Instance	vCPU	Runs
lk-sip	c7g.xlarge	4	livekit-sip (SIP :5060, RTP 30000-30999)
lk-sfu	c7g.xlarge	4	livekit-server (SFU) + Redis
lk-agents-1 … N	c7g.xlarge	4 each	one livekit-agents worker per node

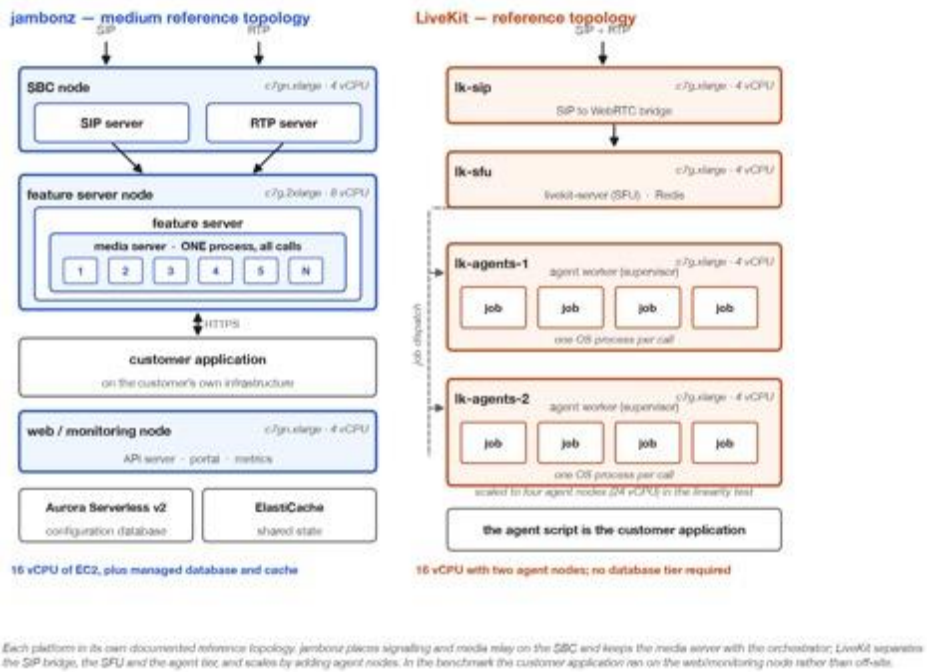
Table 5. LiveKit clustered topology. The 4 vCPU / 8 GB agent node matches the unit LiveKit's own documentation sizes at 10-25 concurrent jobs. Tested with N=2 (16 vCPU total) and N=4 (24 vCPU total).

4.3.2 jambonz clustered topology

The equivalent jambonz deployment is its standard "medium" reference architecture, deployed from the project's own published CloudFormation template with only the modifications needed to place it in the existing test VPC. It splits the media plane differently from the single-node case: the RTP proxy moves to the SBC tier while the media server stays with the orchestrator.

Node	Instance	vCPU	Runs
SBC	c7gn.xlarge	4	drachtio (SIP) + RTP proxy + sidecars
Feature server	c7g.2xlarge	8	feature-server (8 instances) + jambonz media server
Web / monitoring	c7gn.xlarge	4	API server, portal, metrics, and the customer application
Managed services	Aurora Serverless v2 + ElastiCache	n/a	configuration database and shared state

Table 6. jambonz clustered topology (16 vCPU of EC2). Note the managed database and cache tier, which LiveKit's deployment does not require, an asymmetry disclosed in §7.



Each platform in its own documented reference topology: jambonz places signaling and media relay on the SBC and keeps the media server with the orchestrator; LiveKit separates the SIP bridge, the SFU and the agent tier and scales by adding agent nodes. In the benchmark the customer application ran on the web/monitoring node rather than off-site.

Figure 3. Clustered reference topologies, with per-node instance types and vCPU counts. jambonz requires a managed database and cache tier that LiveKit's deployment does not.

4.4 Configuration matrix

ID	Platform	Topology	Turn detection	vCPU
JZ-1	jambonz	single node	native acoustic (on-box)	8
JZ-2	jambonz	single node, single-process app	Deepgram Flux (off-box)	8
JZ-3	jambonz	single node, clustered app	Deepgram Flux (off-box)	8
JZ-4	jambonz	single node, clustered + trimmed app	Deepgram Flux (off-box)	8
JZ-C	jambonz	clustered (medium reference)	Deepgram Flux (off-box)	16
LK-1	LiveKit	single node, default worker	native semantic (on-box)	8
LK-2	LiveKit	single node, tuned worker	native semantic (on-box)	8
LK-3	LiveKit	single node	Deepgram Flux (off-box)	8
LK-4	LiveKit	single node, no local VAD	Deepgram Flux (off-box)	8
LK-C	LiveKit	cluster, 2 agent servers	native semantic (on-box)	16
LK-CF	LiveKit	cluster, 2 agent servers, no local VAD	Deepgram Flux (off-box)	16
LK-4N	LiveKit	cluster, 4 agent servers, no local VAD	Deepgram Flux (off-box)	24

Table 7. All measured configurations. JZ-2 is reported but excluded from capacity conclusions: its ceiling was set by the customer application, not the platform (§7).

5. Results

Observations are presented first and interpreted afterwards.

5.1 Single node, identical hardware

Both platforms on one c7g.2xlarge (8 vCPU), with identical off-box turn detection, so that neither performs turn-detection inference locally.

jambonz (JZ-4)

Concurrency	Completed	Failed	Failure rate	p50 (ms)	p95 (ms)
225	1,800	0	0.0%	1,460	1,580
250	1,997	3	0.15%	1,469	1,590
275	2,069	131	5.95%	1,502	1,633
300	2,048	352	14.67%	1,524	1,682
325	1,919	681	26.19%	1,569	1,783

Table 8. jambonz single-node ladder with off-box turn detection. Sustainable capacity 250. Latency is flat through the clean region and degrades only as the platform saturates.

LiveKit (LK-4, its best single-node configuration)

Concurrency	Completed	Failed	Failure rate	p50 (ms)	p95 (ms)
25	200	0	0.0%	1,558	1,751
50	166	234	58.5%	1,559	1,768

Table 9. LiveKit single-node ladder, off-box turn detection with local voice-activity detection removed — the configuration its community recommended. Sustainable capacity 25.

On the same instance type, with the same workload and the same mocked vendors, the two platforms differ by a factor of ten in sustainable capacity: 250 against 25.

On the same 8-vCPU server, jambonz carried 250 concurrent calls where LiveKit carried 25, a tenfold difference at the same acceptance criteria.

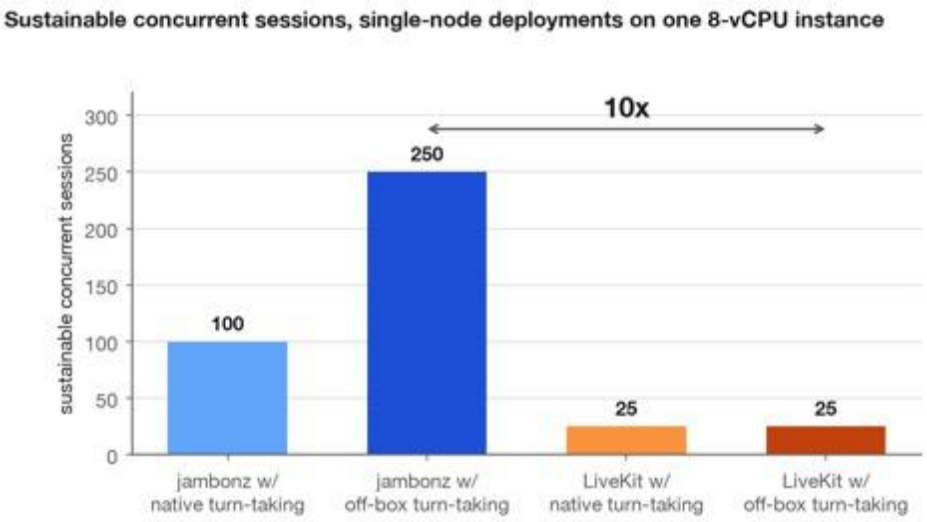


Figure 4. Sustainable capacity on one 8-vCPU instance, at the quality threshold of §3.1. Platform is encoded by hue and turn-taking mode by shade. Speech and language-model vendors are mocked identically for both platforms (see the note below and §3.3).

What the two turn-taking modes mean, and how the speech service was mocked

Native turn-taking means the platform decides for itself when the caller has stopped speaking, using a detector that runs on the platform's own hardware and consumes its CPU. The two platforms do this differently by design: jambonz uses the Krisp turn-taking model inside its media process, while LiveKit runs a transformer-based semantic turn detector in its agent runtime. This is the configuration each platform recommends, and the one a normal deployment would use.

Off-box turn-taking means neither platform runs a detector at all. The speech-to-text service emits explicit turn events (start of turn, interim updates, a provisional end-of-turn, and a final end-of-turn) and the platform simply acts on them as they arrive. In this mode no turn-detection inference runs on the platform's hardware, which is what makes the two platforms directly comparable: the workload is identical and the only difference left is how each one moves audio and manages sessions. Deepgram and AssemblyAI are examples of STT vendors that provide integrated turn-taking models.

The speech-to-text service is mocked, and no traffic leaves the test network. A single mock service runs on a separate host and implements the streaming speech-to-text protocol with fixed, injected latencies, identical for both platforms. It is audio-aware rather than a simple script player: it detects the caller's voiced spans in the arriving RTP and paces scripted transcripts against them, so interim results, the two mid-utterance hesitation pauses, and the turn events all fire in correct relation to the actual audio. Turn events are emitted 350 ms after speech ends, with a provisional end-of-turn 200 ms earlier so that a platform supporting it can begin the language-model request before the turn is confirmed. Both platforms receive byte-identical caller audio and identical vendor timing; no speech vendor is contacted and no vendor charges are incurred. Where a vendor name appears in configuration it selects a *protocol dialect*, not a hosted service (§3.3).

5.2 The turn-detection control

§2.4 predicted that LiveKit's ceiling would not respond to moving turn detection off the box. It did not.

Platform	Native turn detection	Off-box turn detection	Capacity change
jambonz	~100 sessions	250 sessions	+150%
LiveKit	~25 sessions	~25 sessions	none

Table 10. Effect of removing local turn-detection inference. jambonz gains substantially; LiveKit does not move.

The asymmetry is informative. jambonz's native acoustic turn detection runs inside the shared media process, so its cost is charged directly against the resource that limits jambonz. Removing it more than doubles capacity. LiveKit's ceiling is unmoved because the turn detector was never the binding constraint; the per-call process itself is.

LiveKit's latency, by contrast, improved dramatically: p95 turn latency fell from roughly 4.5 seconds with its native semantic detector to roughly 1.75 seconds with off-box turn events. This is a significant finding in LiveKit's favour on the latency axis, one that a user can act on immediately.

We also tested the final step the community recommended: removing local voice-activity detection entirely, which is possible once the STT service supplies turn events. On the single node this changed nothing (25 sessions either way). On dedicated agent nodes it did help, for a reason worth recording in §5.4.

5.3 Clustered deployments

The clustered results are presented as replication rather than escalation. The question is whether the per-vCPU efficiency gap is a property of the architecture or an artifact of one deployment shape.

LiveKit, its reference topology with every recommendation applied (LK-CF)

Concurrency	Completed	Failed	Failure rate	p95 (ms)
25	200	0	0.0%	1,765
30	240	0	0.0%	1,833
35	278	2	0.71%	1,772
40	263	57	17.81%	1,836
45	324	36	10.0%	1,895
50	291	109	27.25%	2,043

Table 11. LiveKit on 16 vCPU (two dedicated agent servers), off-box turn detection, no local VAD, default worker settings. Sustainable capacity 35. Above 35 the region is unstable rather than cleanly degrading — see §7.

jambonz, medium reference topology (JZ-C)

Concurrency	Completed	Failed	Failure rate	p50 (ms)	p95 (ms)	p95 vs baseline
25	200	0	0.0%	1,461	1,582	—
50	400	0	0.0%	1,453	1,577	—
100	800	0	0.0%	1,453	1,573	—
150	1,200	0	0.0%	1,453	1,572	—

200	1,600	0	0.0%	1,452	1,571	baseline
250	2,000	0	0.0%	1,454	1,572	+0.1%
300	2,400	0	0.0%	1,460	1,580	+0.6%
400	3,200	0	0.0%	1,659	1,832	+16.6%
450	3,598	2	0.06%	1,697	1,941	+23.6%
475	3,795	5	0.13%	1,734	2,107	+34.1%
500	3,995	5	0.12%	1,764	2,257	+43.7%
525	4,185	15	0.36%	1,889	2,596	+65.2%
550	4,369	31	0.70%	1,961	2,530	+61.0%
600	3,091	1,709	35.6%	2,112	2,751	+75.1%

Table 12. jambonz on 16 vCPU of EC2 plus managed database and cache. Latency is flat across a twelvefold concurrency range, from 25 to 300 sessions, then rises while the failure rate stays near zero — the behaviour analysed in §5.6. The ladder was run to its safety-stop threshold of 25% failures, reached at 600 sessions.

Sustainable capacity is 450 sessions. 0.06% failures and p95 latency 23.6% above the unloaded baseline, satisfying both clauses of the capacity threshold. Applying only the failure criterion would give 525, at which point p95 has risen 65% and callers are waiting appreciably longer for every reply. We publish the lower, latency-constrained figure.

Sensitivity of the clustered figure to the latency threshold

Because the 25% latency clause is a judgement rather than a measurement, and because it is the one methodological choice in this paper that materially moves one of our own numbers, we state what other choices would have produced:

Latency threshold (p95 above unloaded baseline)	Resulting capacity
No latency clause — failure rate only	525
+45%	500
+35%	475
+25% — the threshold used in this paper	450
+20%	400
+15%	300

Table 13. Clustered jambonz capacity as a function of the latency criterion. The figure is 450 for any threshold between roughly 24% and 34%; a tighter threshold of 20% would give 400, and a looser threshold of 35% would give 475.

A reader who prefers a different threshold can therefore read their own figure directly from the ladder in Table 12. Our chosen threshold sits near the lower edge of the band that yields 450, so the published figure is conservative with respect to that choice rather than flattering to it. The ladder was run at

25-session granularity through this region so that the sensitivity could be stated from measurement rather than interpolation.

5.4 Horizontal scaling

Doubling LiveKit's agent tier from two nodes to four doubled its capacity precisely, from 35 to 70 sessions.

Agent servers	Total vCPU	Capacity	Sessions per agent server	CPU cores per session
2	16	35	17.5	0.100 – 0.116
4	24	70	17.5	0.105 – 0.112

Table 14. LiveKit horizontal scaling. Per-node capacity and per-session CPU cost are unchanged — scaling is linear and predictable.

This deserves to be stated generously, because it is a true strength. LiveKit scales horizontally and evenly. At 70 concurrent sessions the four agent workers sat at 48%, 49%, 48% and 49% CPU respectively. Job dispatch is well balanced, which is precisely why the scaling is linear. An operator can size a LiveKit fleet with confidence by budgeting roughly 17.5 sessions per four-vCPU agent server and adding nodes.

The finding is therefore not that LiveKit fails to scale. Instead, it is that **the per-session cost being replicated is high**, so the hardware required to reach a given call volume is correspondingly large.

This experiment also explained an earlier null result. Removing local voice-activity detection did nothing on the single node but helped on dedicated agent nodes (30 to 35 sessions), because on the single node the agent processes were also contending with the SIP tier's packet-handling work; only once the agent tier is isolated does per-session agent CPU become the sole constraint, at which point shaving it converts directly into capacity.

5.5 Mechanism: where the capacity goes

Per-process sampling allows the capacity difference to be attributed rather than merely observed.

jambonz at its knee (single node, 8 vCPU)

Concurrency	Box CPU (mean)	Box CPU (p95)	Load avg	Media server	Customer app	Orchestrator	Memory
250	52%	95%	5.3	46%	2.8%	2.0%	12.5 GB
275	57%	99%	7.0	47%	3.0%	2.5%	12.6 GB
300	60%	99%	7.0	54%	3.5%	3.0%	12.8 GB

Table 15. jambonz per-component CPU as a percentage of the whole 8-core machine. The shared media process is the dominant consumer and the binding constraint.

LiveKit at its knee (cluster, per 4-vCPU node)

Concurrency	SIP node	SFU node	Agent node 1	Agent node 2	Agent load avg	Agent cores	Cores/session
25	48%	6%	35%	37%	1.9 / 1.9	2.86	0.115
30	50%	8%	45%	42%	2.5 / 2.2	3.45	0.115
35	52%	8%	49%	51%	2.5 / 2.9	4.07	0.116
45	56%	11%	59%	59%	4.3 / 4.6	4.77	0.106

Table 16. LiveKit per-node CPU as a percentage of that node's own 4 cores. Agent nodes saturate; the SFU is nearly idle.

Three conclusions follow.

- **Per-session cost differs by roughly eight times:** LiveKit consumes 0.110 CPU cores per concurrent session inside its agent processes, measured across six load levels on each of two cluster sizes (range 0.100 to 0.116). jambonz's shared media process consumes 0.014 cores per session, measured across twelve concurrency steps from 25 to 550 sessions (range 0.0129 to 0.0157). Both figures are near-constant with load: the platforms differ in the size of a per-session cost, not in how that cost scales. This is the hypothesis of §2.4 confirmed.
- **The SFU is not the expensive component:** LiveKit's SFU never exceeded 17% of a single four-vCPU node. For two-participant audio-only sessions it is close to free. The cost is the agent runtime, not WebRTC media forwarding. This is a point we want to make clear, as it would be easy to misread this paper as a criticism of the SFU.
- **Neither platform is memory-bound:** jambonz peaked at 12.8 GB of 15.3 GB with no swapping; LiveKit's agent nodes used 3.4 to 4.5 GB of 8 GB. Capacity is a CPU and scheduling story on both platforms.

The reason is per-call cost. **Every concurrent session costs LiveKit about eight times more CPU than it costs jambonz.**

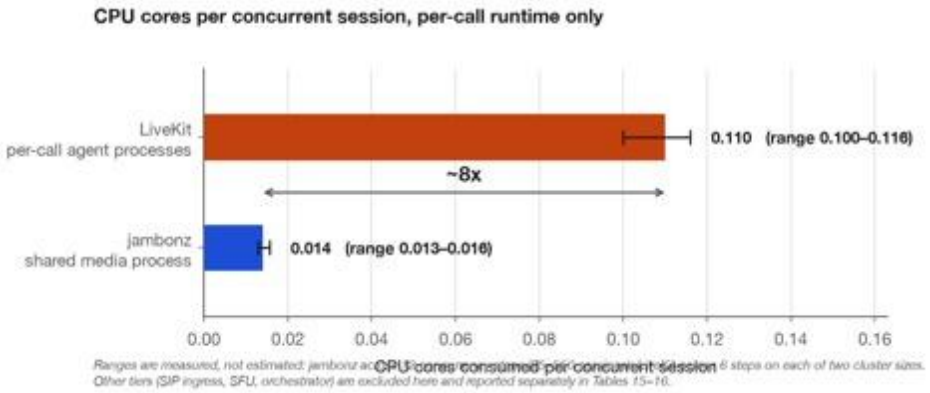


Figure 5. CPU cores consumed per concurrent session by each platform's per-call runtime. Ranges are measured across every load level tested, not estimated. Thresholds compare LiveKit's per-call agent processes with jambonz's shared media process.

What this figure does and does not compare

It isolates the *per-call runtime* on each platform (LiveKit's agent processes against jambonz's media process) because that is the component the two architectures implement differently, and therefore the one the paper's argument turns on. It deliberately excludes LiveKit's SIP and SFU tiers and jambonz's orchestrator, all of which carry real costs that Tables 15 and 16 report separately.

It follows that the eight-fold figure here and the tenfold capacity ratio of §5.1 are two different measurements rather than the same result stated twice, and readers should not expect them to reconcile arithmetically. Sustainable capacity depends on the whole deployment and on which tier's failure mode binds first, which, as §5.6 shows, is not the same tier in every topology. The per-session runtime cost explains the direction and rough magnitude of the capacity difference; it does not fully determine it.

5.6 Why does latency degrade before calls fail?

The clustered jambonz deployment degrades in an unusual and instructive way. Response latency rises substantially while the failure rate stays near zero. Because this is the observation that forced the latency clause in our success threshold (§3.1), it is worth establishing what causes it rather than merely noting it.

Concurrency	Media node CPU	Media node load (8 cores)	Media process	Every other tier	p50 (ms)	p99 − p50 (ms)
300	55.2%	5.08	53.2%	≤ 13.8%	1,460	143
400	74.8%	8.92	70.0%	≤ 15.4%	1,659	422
500	91.9%	15.04	86.3%	≤ 21.4%	1,764	1,095
550	94.1%	16.26	88.7%	≤ 23.9%	1,961	1,142

Table 17. Clustered jambonz: resource state against measured latency, per ladder step, ramp-up excluded. "Every other tier" is the highest of the SIP, customer-application, and load-generator nodes. Load average is on an 8-core node, so values above 8 indicate a run queue deeper than the available cores.

Four observations identify the mechanism.

- **The onset coincides with the run queue exceeding the core count:** Latency is flat at approximately 1,453 ms across a twelvefold range of concurrency, from 25 through 300 sessions, while the media node's load average stays below its eight cores. It departs from that baseline between 300 and 400, exactly where load average crosses from 5.08 to 8.92.
- **The media process is the load:** It accounts for 86 to 89% of the whole eight-core node at the upper steps, roughly seven of eight cores. The orchestrator, by contrast, never exceeds 5%, so the delay is not in call control or application logic.
- **No other tier is close to saturation:** At the point where latency has risen 35%, the SIP tier is at 4.8%, the customer-application tier at 10.1%, the mocked vendor host at 10.9%, and the load

generator at 23.9%. This excludes the additional network hop, the customer application, the mocks, and the test rig as explanations.

- **The dispersion signature is queueing, not a fixed offset:** The spread between p99 and p50 grows from 143 ms to over 1,100 ms. A longer media path would displace the entire distribution uniformly; a saturated run queue delays some sessions far more than others, which is what the data shows.

The mechanism is therefore **scheduling delay in timer-driven media work.** Audio generation and RTP pacing are soft-real-time activities: they are driven by timers rather than by request arrival. When the run queue is deeper than the available cores, those timers fire late, and the agent's first audio frame is simply emitted later. Nothing times out, so nothing fails, which is precisely why the failure rate stays near zero while latency climbs.

Calls only begin to fail at 550 sessions, once the accumulated delay starts to exceed the per-turn timeout.

Why does the single node fail differently?

This also explains a contrast that would otherwise look inconsistent. The single-node deployment began failing at 275 sessions while its CPU was only 57%. That is, it failed well before saturation. There, all five tiers contend for the same eight cores, and the failure arises from contention between them rather than from media scheduling delay. The clustered deployment, with the media tier isolated on its own node, instead runs to 91.9% CPU with a 0.12% failure rate and degrades smoothly.

Isolating the media tier **converts a hard failure mode into a soft latency degradation.** That is an operational benefit of the clustered topology. It is also a **warning**. An operator watching only call-completion rates would see a clustered deployment as healthy at a concurrency where callers are in fact waiting noticeably longer for replies. Latency, not failure rate, is the leading indicator on this topology.

How firmly is the mechanism established?

The location of the delay is an **inference by elimination**, not a direct measurement, and we label it as such. What is measured is unambiguous. The media node is the only tier under pressure, the onset of latency growth coincides with its run queue exceeding its core count, every other tier remains below a quarter of its capacity, and the dispersion signature is characteristic of queueing rather than of a fixed additional delay.

What we have not done is instrument the interval between the mocked vendor emitting audio and that audio reaching the caller, which is the measurement that would locate the delay directly rather than by exclusion.

Readers should therefore treat the observation (that latency degrades substantially while calls still complete) as measured fact, and the attribution of that delay to media scheduling as a well-supported hypothesis consistent with all available data. The practical guidance in this section does not depend on

the mechanism being correct. The two-part performance threshold and the recommendation to monitor latency rather than failure rate follow from the observation alone.

The two platforms fail differently: at its limit LiveKit rejects calls, while jambonz keeps completing them more slowly.

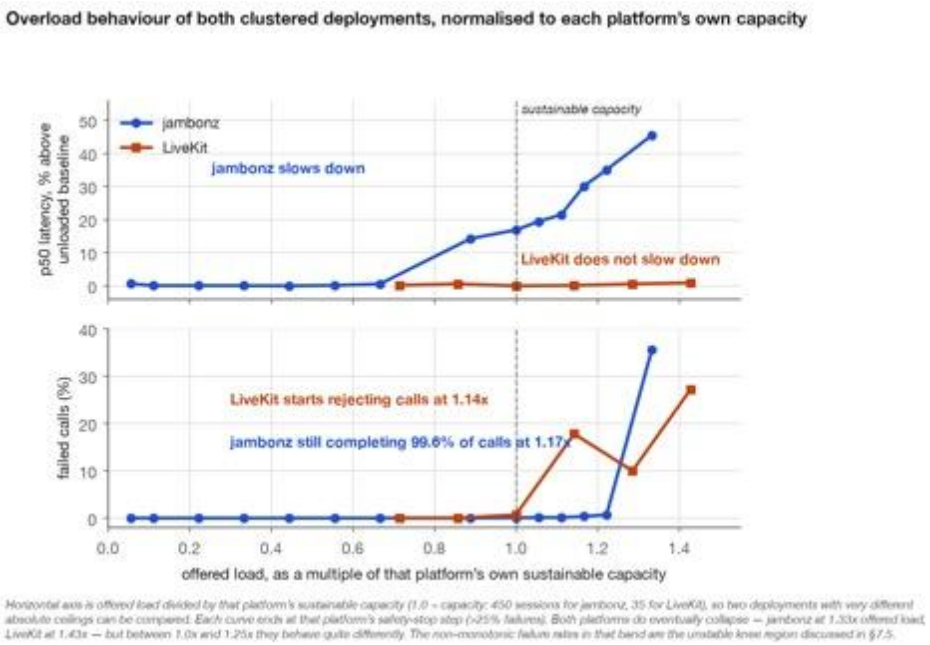


Figure 6. Overload behaviour of both clustered deployments. The horizontal axis is offered load divided by that platform's own sustainable capacity, so deployments with very different absolute ceilings can be compared. Each curve ends at that platform's safety-stop step.

This is a difference in overload behaviour, not in headroom. LiveKit's admission control holds response time essentially constant (its median latency varies by 14 ms across failure rates from zero to 27%) and sheds load by refusing calls. jambonz accepts the offered load and response times lengthen instead, completing 99.6% of calls at 1.17 times its capacity where LiveKit is already rejecting 17.8%. Both platforms do eventually collapse, jambonz at 1.33 times capacity and LiveKit at 1.43.

Note that **neither behaviour is simply better.** An operator holding a strict response-time SLA may prefer admission control, which keeps every admitted call fast and makes the failure visible immediately at the edge. An operator who would rather make a caller wait than turn them away will prefer graceful degradation.

The practical consequence is a monitoring one: on jambonz, latency is the leading indicator of saturation and call-completion rate is a lagging one, so a dashboard watching only completion rates would report a healthy platform at a concurrency where callers are noticeably waiting.

It also explains why the standard of §3.1 needs its latency clause, and why that clause changes only one figure in this paper. A latency criterion can never bind on LiveKit, because LiveKit converts overload into rejection rather than delay. On jambonz it binds, because jambonz converts overload into delay.

5.7 Latency

At matched load (25 concurrent) with identical off-box turn detection:

	Turn samples	Mean (ms)	Median (ms)	SD (ms)	p95 (ms)
jambonz	1,400	1,453	1,460	89	1,580
LiveKit	1,365	1,563	1,556	110	1,765

Table 18. Turn latency at matched concurrency, identical turn detection.

Treating each call's mean as the unit of analysis, the difference is −96 ms (bootstrap 95% CI [−99, −93] ms), Mann-Whitney $p \approx 6\times10^{-63}$, Cliff's delta −0.974 — that is, roughly 97% of jambonz calls were faster than 97% of LiveKit calls. Every one of the seven turn positions in the script showed the same direction and a similar magnitude.

The p-value is not the interesting statistic here; with this sample size and these tight distributions almost any difference would be "significant". What makes the result robust is replication:

Platform	Independent runs	Clean ladder steps	p50 range (ms)	Mean	SD
jambonz	3	10	1,450 – 1,469	1,456	6.0
LiveKit	3	8	1,551 – 1,561	1,554	3.5

Table 19. Median turn latency across independent runs. The ranges do not overlap, and LiveKit's figure barely moves across three different topologies (single node, two-node cluster, four-node cluster).

The honest framing is that this is a modest but highly reproducible advantage. About 700 ms of the roughly 1,500 ms end-to-end figure is irreducible mocked-vendor and protocol latency common to both platforms; netting that out, the platform-attributable difference is approximately 13%. A plausible mechanism is the additional media hops in LiveKit's path (SIP to the bridge, WebRTC to the SFU, WebRTC again to the agent) each with its own jitter buffer, although we did not instrument the internal hops to confirm this.

Your turn-taking choice moves response time more than your platform choice does. The larger improvement is available to LiveKit users.

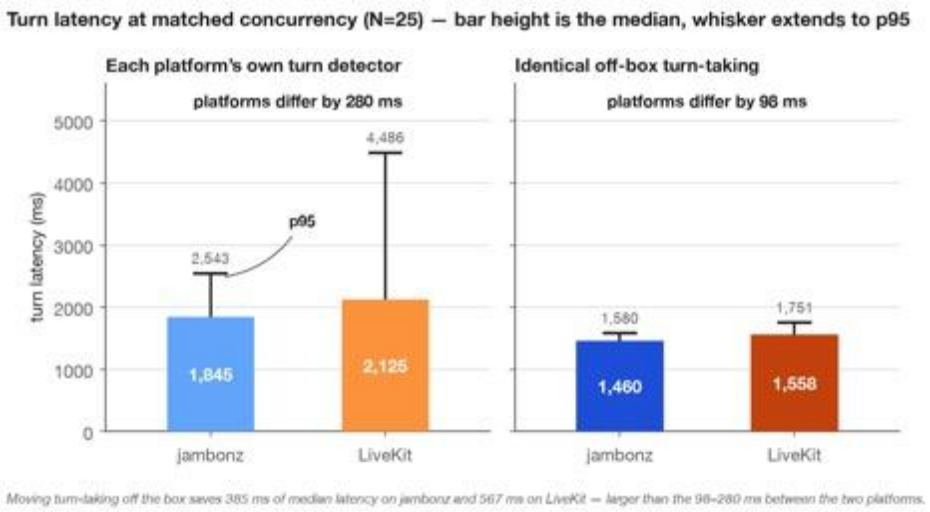


Figure 7. Turn latency at matched concurrency (N=25). Both panels share one vertical scale so they can be read against each other. Single-run figures shown for illustration; the statistical treatment is in Tables 18 and 19.

Two readings of this figure, and **one caution**. The left panel shows that each platform's own turn detector dominates its response time, and that LiveKit's costs it considerably more: a p95 of 4,486 ms against 2,543 ms. The right panel shows that once turn-taking is identical and off-box, LiveKit's p95 more than halves to 1,751 ms and the median gap between the platforms narrows to 98 ms. **The latency available to LiveKit from moving turn-taking off the box is therefore larger than anything we found in jambonz's favour on this axis**. It is actionable today by any LiveKit user.

The caution is against **over-reading the 98 ms**. It is roughly 6% of end-to-end response time, and approximately 700 ms of the 1,500 ms total is irreducible mocked-vendor and protocol latency common to both platforms; netting that out gives a platform-attributable difference of about 13%. The figure illustrates a single run at one concurrency level. What makes the finding robust is not this chart but the replication recorded in Table 19. Non-overlapping per-run ranges across six independent runs and three LiveKit topologies.

A note on what "off-box turn-taking is faster" does and does not establish:

We measured turn-taking latency, not turn-taking accuracy. The two mechanisms are not doing the same work. A semantic turn detector reasons about whether an utterance is *complete*, not merely whether audio has stopped, and it is reasonable to expect that reasoning to cost time. Our workload contains two scripted hesitation pauses, which both platforms handled correctly in every configuration, but scripted hesitations are a gentler test than real speech: callers trail off mid-thought, pause to choose a word, and end sentences ambiguously. A semantic model may ride out those cases where a threshold on a speech-to-text stream fires early and interrupts the caller.

We have no data either way, because interruption accuracy was never a measured outcome of this study. **A slower detector may therefore be buying something this benchmark did not evaluate.** Readers should take the latency figures above as what they are (a measurement of response time under one scripted workload) and should not read them as a recommendation to prefer one turn-detection mechanism over another on quality grounds. Establishing that would require a different experiment, with adversarial turn boundaries and a measure of false interruptions.

5.8 Cost, priced once

Applying the §3.8 derivation to on-demand list prices for the instance types used:

	Sessions/vCPU	vCPU for 1,000 sessions	Cost/hour	Cost/month
jambonz	31.3	32	$1.16	~$850
LiveKit (best case)	3.1	320	$11.60	~$8,500

Table 20. Infrastructure cost for 1,000 concurrent sessions. AWS us-west-2 on-demand, Graviton c7g at $0.03625 per vCPU-hour, 730-hour month, priced [INSERT DATE]. LiveKit is costed using its single-node figure, which is its most favourable, since its clustered configurations carry underutilised SIP and SFU nodes.

This covers the infrastructure tier only. Speech and language-model vendor charges are identical for both platforms and in production typically exceed infrastructure cost by a wide margin. The correct reading of Table 20 is therefore a ten-fold difference in the portion of the bill that platform choice controls, not a ten-fold difference in total cost of ownership. Reserved or spot pricing reduces both figures proportionally and does not change the ratio.

6. Fairness: What We Did to Make LiveKit Fast

A benchmark published by one of the vendors being compared carries an obvious credibility problem. Our mitigation is to make the competitor as fast as we could, to document every step, and to invite correction. This section is that record.

6.1 Configuration defects we found and fixed

LiveKit's first measured ceiling was approximately **14 sessions**, and the cause was a configuration issue rather than a capacity limit: its SIP bridge attempted ICE negotiation against public STUN servers to establish what was, in our deployment, a loopback media path. Those queries timed out under concurrency and calls were rejected. Enabling ICE-lite so that only host candidates are offered removed the problem entirely. Every LiveKit figure in this paper is measured with that fix in place.

This is worth stating plainly because a less careful benchmark would have published the 14.

6.2 Tuning, and reverting tuning

We raised LiveKit's worker admission threshold from its default, on the theory that the default was throttling before the hardware was full. It was not: opening the threshold caused the worker to accept work past saturation, converting clean rejections into call-setup timeouts, with no capacity gain. We reverted to the default and reported the default configuration, because it is better calibrated. LiveKit's defaults are well chosen for this workload.

6.3 Community consultation

We posted our hardware, versions, configuration and observed numbers to the LiveKit community forum and asked what we had done wrong. The responses were substantive and we acted on all of them:

- Keep the process-based job executor rather than switching to threads. Threading would serialise per-call inference behind Python's global interpreter lock. We did not switch.
- Return the worker admission threshold and process pre-warming to their defaults. We did, and confirmed the advice was correct.
- Note that the semantic turn detector already runs in a shared inference process, so it is not a per-call cost; local voice-activity detection is the only per-job inference. We verified this in the installed source and then removed the local VAD.
- Move agent workers off the SFU host. Described as the single largest available lever. We rebuilt into the tiered topology, and then doubled the agent tier to test whether scaling was linear.

Their own documentation corroborates the result. LiveKit's deployment guide sizes an agent server of exactly the shape we used (four cores, 8 GB) at "10-25 concurrent jobs." Community respondents confirmed that our figure "matches LiveKit's own sizing, not a misconfig". The capacity number in this paper is therefore not in dispute between us and the vendor; only its implications are.

6.4 Where our own configurations were imperfect

For symmetry, the same scrutiny applied to jambonz surfaced problems on our side, disclosed here and in §7: an early acoustic turn-detection run used a threshold lower than intended; the single-node and clustered jambonz deployments differ in one respect beyond topology; and our own customer-tier application became a bottleneck before the platform did and had to be fixed twice.

None of these flatter us. All are recorded.

7. Discussion and Threats to Validity

Naming the limits of a benchmark is what distinguishes it from marketing. The following are the material threats to the validity of these results, in roughly descending order of importance.

7.1 The cost claim covers infrastructure only

Speech-to-text, language-model and text-to-speech charges dominate the real operating cost of a production voice agent, and they are identical across both platforms. The ten-fold infrastructure difference in §5.8 applies to a minority of the total bill. Readers sizing a business case should model vendor spend first.

7.2 Mocked vendors

Using mocks is the central methodological choice, and it cuts both ways. It removes vendor latency variance and cost, making the platform comparison clean and repeatable. **That is the point.** But it means absolute latency figures are not representative of production, and only differences between platforms are meaningful. It also means our mocks' behaviour under concurrency, rather than a real vendor's, is part of the measured system.

7.3 The customer application tier is part of the system

The webhook application that supplies call instructions is customer code, not platform code, and in our first clustered runs it (not jambonz) set the ceiling.

A single Node process could not complete WebSocket handshakes fast enough at 150 concurrent calls. Running it as four clustered instances raised the clean point to 200, and removing per-turn event logging raised it to 250. We report 250 and disclose the intervention.

Two observations follow:

1. This tier is part of a deployment's capacity
2. Comparisons of this kind must account for it explicitly

7.4 Asymmetries between our own jambonz configurations

- **Managed services:** The clustered jambonz deployment includes a managed database and cache tier beyond its EC2 nodes; the LiveKit cluster requires only a Redis process. The "16 vCPU versus 16 vCPU" comparison is therefore EC2-only, and modestly favourable to jambonz on that basis.
- **SIP server build:** The single-node jambonz runs used a SIP server build with licence enforcement compiled out; the clustered deployment used the standard licensed artifact. If the licence path carries any per-call overhead, the two jambonz configurations are not perfectly comparable with each other.

- **Turn-detection threshold:** The native acoustic turn-detection run (JZ-1) executed at a lower confidence threshold than intended, owing to a stale application build. We report it as preliminary; it affects the ~100 figure in Table 10, not the off-box results on which the headline rests.

7.5 The knee is a region, not a point

Above the clean threshold, behaviour is noisy rather than monotonically degrading. In one LiveKit ladder the failure rate at 40 concurrently exceeded that at 45, because a single agent worker experienced a transient load spike.

We report the last clean step as capacity and note that the transition region is unstable. An operator should not plan to run at the knee.

7.6 The latency mechanism is inferred, not directly measured

Section 5.6 attributes the clustered deployment's latency degradation to scheduling delay in timer-driven media work. That attribution rests on eliminating every other candidate tier and on the shape of the latency distribution, not on instrumenting the media path itself. It is the least directly evidenced claim in this paper, and a reader inclined to challenge one thing should challenge this.

The underlying observation (that p95 latency rises more than 40% while the failure rate remains near zero) is measured directly and does not depend on the explanation being right.

7.7 Generalisation

Consistency across eleven configurations is strong evidence that the efficiency difference is a property of the architectures rather than of any single deployment. But we tested two products, one workload family, one codec, one instance family, in one region. We do not claim the ratio is a constant, and we would expect it to move with workload shape. In particular, workloads with heavier local inference, or with multi-party media, would change the picture.

7.8 Workloads not covered

This benchmark is narrow. The following are outside its scope, and the reasons should be read as boundaries on the claim rather than as gaps to be filled by inference.

- **Video:** The workload is audio-only telephony; nothing here measures video encoding, transport, or SFU forwarding cost, so no conclusion should be drawn about either platform's video performance.
- **Multi-party rooms:** Every session in this study is a single caller and a single agent; readers should not assume the capacity ratio holds for conference-style or multi-participant deployments, where LiveKit's SFU architecture is designed to do different work.

- **Browser/WebRTC-originated sessions:** All calls entered through SIP, which is jambonz's native path and requires a bridging hop on LiveKit; a browser-originated session on LiveKit skips that bridge entirely, and its relative performance is not something this data speaks to.
- **Outbound campaigns:** The workload is a single scripted inbound conversation per session, not the batch dialing, pacing, or answering-machine-detection patterns typical of outbound campaigns, so no claim is made about outbound-specific throughput or tooling.
- **Recording/egress-heavy workloads:** Neither platform was tested with call recording, transcription export, or media egress enabled, and readers should not assume the measured CPU costs would hold once those additional per-call operations are added.

7.9 Right of reply

We invite LiveKit to review our configuration and methodology, tell us what we have wrong, and publish corrected numbers. The harness, configurations and data are public precisely so that this is possible, and we undertake to re-run the benchmark and update this paper in response to any substantive correction.

8. Conclusions

Under an eight-turn conversational telephone workload, with identical mocked vendors and identical off-box turn detection, jambonz sustained approximately 250 concurrent voice-agent sessions on an 8-vCPU instance, compared to LiveKit's 25. That's roughly 31 sessions per vCPU against 3. Both figures are measured at the same standard defined in §3.1: fewer than 0.5% failed calls and p95 turn latency within 25% of baseline.

The result is robust for three reasons rather than one. It reproduced across eleven configurations, including four LiveKit variants on a single node and two cluster shapes, and LiveKit's per-vCPU efficiency never left the range 2.2 to 3.1. It survived the removal of the most obvious confound, turn detection, which we moved off-box on both platforms. And it is explained by a measured mechanism: approximately 0.11 CPU cores per session inside a per-call agent process against approximately 0.014 cores per session in a shared media process.

The difference is architectural. A process-per-call runtime buys fault isolation and sidesteps interpreter-level concurrency limits, at the cost of paying a fixed per-session overhead that a shared media plane amortises. Neither choice is wrong in the abstract; they are optimised for different problems, and LiveKit's design reflects its origins in browser-based multi-party media rather than high-density telephony.

LiveKit scales horizontally and predictably. We measured exactly linear scaling and evenly balanced dispatch. As such, the practical consequence is not that a given call volume is unreachable, but that reaching it requires roughly an order of magnitude more compute. For teams self-hosting telephone voice agents at scale, that is the portion of the infrastructure bill that platform choice controls.

8.1 Choosing between them

The recommendation that follows from this data is a **threshold.** For deployments in the range of tens of concurrent calls, and, on our measurements, up to around a hundred, LiveKit is a sound choice. Capacity is simply not the constraint that should decide it. At that scale the platform costs a handful of small nodes either way. The decision should turn on **features**: whether the application needs video, multi-party rooms, browser and mobile clients, or a managed cloud option, all of which LiveKit offers and jambonz does not.

If sustained concurrency runs to the high hundreds or thousands**, the architecture becomes the deciding factor.** A tenfold difference in sessions per vCPU is the difference between four servers and forty, with the deployment, monitoring, failure-domain and capacity-planning burden that implies. Teams sizing for that range should evaluate jambonz, and should test it on their own workload rather than taking our figures on trust, which is why the harness is published.

8.2 Considerations beyond capacity

Two properties of jambonz bear on the same decision but were not measured by this benchmark, and we flag them as context rather than as findings.

- **SIP and telephony integration:** jambonz is SIP-native by design rather than by extension, and interoperates directly with carrier SBCs, SIP trunking providers, and on-premise PBX systems of essentially any description. For deployments whose voice agents must sit inside an existing telephony estate (alongside a PBX, behind a carrier trunk, or bridging both) that integration surface is the platform's original purpose rather than a gateway bolted onto a WebRTC core.
- **Call-handling features:** jambonz also provides the call-control machinery that contact-centre applications typically require: queueing, blind and attended transfer, conferencing, recording, and supervisor functions such as monitoring and barge-in. Applications of that kind often need these long before they need high concurrency.

Neither point was exercised by this workload, and neither bears on the capacity measurements. Readers should evaluate both on their own merits rather than on this paper's authority. We mention them only because a reader choosing a platform for telephony-heavy or contact-centre work is unlikely to be deciding on session density alone.

If you are running self-hosted voice AI agents at meaningful scale, or planning to, the numbers in this paper are worth checking against your own workload rather than taking on trust. Capacity claims are cheap to write and expensive to verify, which is exactly why we published the harness instead of just the conclusion. Run it yourself, change what you disagree with, and publish what you find. We mean the invitation in §7.9 literally. If LiveKit or anyone else finds an error in our methodology, tell us and we will re-run the benchmark and correct this paper.

Appendices

Appendix A — Versions tested

Component	Version
livekit-server	1.13.4
livekit-agents	1.6.6 (Python 3.11)
livekit-plugins-deepgram	[INSERT]
livekit-sip	built from source, arm64, commit [INSERT]
lk CLI	2.18
jambonz	11.0.4
Benchmark harness	1.0.0

Table A1. Exact versions. jambonz tags to be cut after the fixes described in Appendix B are released.

Appendix B — Raw data index

Each data folder includes summarized ladder results alongside resource metrics sampled every two seconds for all cluster nodes. Furthermore, the five instances designated by ● provide granular per-step statistics and chronological event logs for individual turns.

Run directory	Configuration
runs/cap1-jambonz ●	JZ-1 — jambonz single node, native turn detection
runs/jz-flux ●	JZ-2 — jambonz single node, single-process app (excluded)
runs/jz-flux-cluster	JZ-3 — jambonz single node, clustered app
runs/jz-flux-fine	JZ-4 — jambonz single node, definitive
(transcribed — see runs/CONFIGURATIONS.md)	JZ-C — jambonz clustered
runs/lk-default ●	LK-1 — LiveKit single node, default worker
runs/lk-tuned ●	LK-2 — LiveKit single node, tuned worker
runs/lk-flux ●	LK-3 — LiveKit single node, off-box turn detection
runs/lk-flux-novad	LK-4 — LiveKit single node, no local VAD
runs/lk-cluster-native, runs/lk-cluster-fine	LK-C — LiveKit cluster, 2 agent servers

runs/lk-cluster-flux	LK-CF — LiveKit cluster, off-box turn detection
runs/lk-cluster-4agents	LK-4N — LiveKit cluster, 4 agent servers

Appendix C — Pricing snapshot

Instance type	vCPU	On-demand $/hour	$/vCPU-hour
c7g.2xlarge	8	$0.2900	$0.03625
c7g.xlarge	4	$0.1450	$0.03625
c7gn.xlarge	4	$0.2496	$0.06240

Table E1. AWS us-west-2 Linux on-demand list prices, retrieved [INSERT DATE].

Appendix D — The community exchange

Full thread: community.livekit.io — tuning recommendations for self-hosted LiveKit SIP

We posted our config and observed ~25-session ceiling to the LiveKit forum. The community confirmed it matched LiveKit's own sizing, not a misconfig, and recommended: keep load_threshold at its default 0.7, stay on the process executor (not threads, to avoid GIL contention with local VAD), reduce num_idle_processes back to ~CPU count, and move agent workers off the SFU/SIP node as the biggest lever for scale. We adopted all of it.

---

[1] https://github.com/jambonz/jambonz-livekit-load-testing-benchmarks